Interactivity-Constrained Server Provisioning in Large-Scale Distributed Virtual Environments

Duong Ta, Thang Nguyen, Suiping Zhou, Xueyan Tang, Wentong Cai, and Rassul Ayani

***Abstract*—Maintaining interactivity is one of the key challenges in distributed virtual environments (DVE), e.g., online games, distributed simulations, etc., due to the large, heterogeneous Internet latencies; and the fact that clients in a DVE are usually geographically separated. In this paper, we consider a new problem, termed the *interactivity-constrained server provisioning* problem, whose goal is to minimize the number of distributed servers needed to achieve a pre-determined level of interactivity. We identify and formulate two variants of this new problem and show that they are both NP-hard via reductions to the set covering problem. We then propose several computationally efficient approximation algorithms for solving the problem. The main algorithms exploit dependencies among distributed servers to make provisioning decisions.**

We conduct extensive experiments to evaluate the performance of the proposed algorithms. More specifically, we use both static Internet latency data available from prior measurements and topology generators, as well as the most recent, dynamic latency data collected via our own large-scale deployment of a DVE performance monitoring system over PlanetLab. The results show that the newly proposed algorithms that take into account inter-server dependencies significantly outperform the well-established set covering algorithm for both problem variants.

I. Introduction

Multi-player online games, distributed military simulations, collaborative design, virtual shopping malls, etc. are typical examples of distributed virtual environments (DVEs) [1]. Essentially, DVEs are distributed systems that enable multiple geographically separated clients to interact with each other in real time within a shared, computer-generated 3D virtual world, where each client is represented by an *avatar*. A client controls the behavior of his/her avatar by various *inputs*, and the *updates* of an avatar's state need to be sent to other clients in the same part of the virtual world. In this way, each client can be aware of and interact with other nearby clients.

Large-scale DVEs with thousands of geographically separated clients interacting concurrently often require a distributed server architecture [2], which may involve multiple servers located in different data centers across the Internet. In such architecture, each client interacts with others through these servers. For better scalability, the common "divide-and-conquer" practice spatially partitions the large virtual world into distinct *zones*, with each zone managed by only one server. Interactions only happen among clients in the same zone, and clients may move from one zone to another. For fast-paced first person shooters (FPS) games, the duration that players stay in a zone may be short, e.g., under one hour. On the other hand, in highly popular massively multi-player role-playing games (MMORPG), players may remain in a zone for quite long, typically on the order of several months.

Maintaining good interactivity for DVEs has been very challenging due to the heterogeneous nature of the Internet and the fact that clients in a DVE are usually geographically distributed. It is likely that a large number of clients in a zone may be far away (in terms of round-trip network latency) to the server hosting that zone, thus the interactivity of the DVE for those clients may be greatly degraded. Previous work [3], [4], [5] have focused on the problem of maximizing interactivity given limited server resource. This problem, referred to as the zone mapping problem in [4], [5], assumes that the underlying infrastructure is given (in terms of the number of distributed servers as well as their locations), which may cause over or under provisioning.

Before deploying any DVEs over the Internet, an important consideration is how much server resource is needed to meet certain interactivity requirement. In this paper, we consider a new problem, referred to as the interactivity-constrained server provisioning problem. Essentially, this problem looks at how to achieve a pre-specified level of interactivity for a certain DVE configuration with minimum number of servers. Solutions to such problem would allow DVE administrators to specify or fine tune the desirable interactivity level of the DVE based on their organizations' financial constraints or other business objectives.

Note that this server provisioning problem complements existing approaches such as the zone mapping approach described above. For example, the DVE administrator may want to guarantee a certain level of interactivity, e.g., every zone in the DVE must have at least 90% of its clients with round-trip client-server latency under 100ms, and at the same time minimize the number of servers. This problem obviously cannot be solved using the zone mapping approach. The main reason is that such approach only maximizes the overall interactivity of the DVE; assuming a fixed number of distributed servers that are already provisioned. For this kind of problem, the DVE administrator may use some server provisioning algorithms (described later in this paper) to provision a minimum number of servers that meet the interactivity requirement. Subsequently, zone mapping approaches can be used to further improve interactivity if needed, based on the already provisioned server infrastructure.

Below, we summarize the primary contributions of this

Duong Ta, Thang Nguyen, Suiping Zhou, Xueyan Tang, Wentong Cai are with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. Email: {binhduong, ttnguyen, asspzhou, asxytang, aswtcai}@ntu.edu.sg.

Rassul Ayani is with the School of Information and Communication Technology, Royal Institute of Technology, Sweden. Email: rassul@imit.kth.se

paper.

- We consider a new problem named interactivity-constrained server provisioning. We investigate and formulate two variants of this problem. We also show that both of these two variants are NP-hard, by reducing each of them to the set covering problem, which is NP-hard.
- We propose several computationally efficient approximation algorithms for the two variants mentioned above. The main algorithms take into account the unique characteristics of the new problem's context and formulation, such as the dependencies among distributed servers.
- We develop and deploy a distributed software framework named DINE for enabling more realistic evaluations of the proposed algorithms under dynamic Internet conditions. The framework can be used as either an evaluation platform for the development of new interactivity enhancement algorithms, or a real-world performance monitoring and management suite for existing DVEs.
- We conduct extensive experiments with realistic models and settings to evaluate the effectiveness of the proposed algorithms. In particular, we use both static Internet latency models generated by the tool DS^2 [6], and BRITE [7], as well as dynamic latency data gathered over a two-week period from PlanetLab using DINE. We also use iPlane [8], an Internet performance estimation system, to collect input data for the algorithms[1]. The main goal of employing iPlane is to evaluate the sensitivity of the proposed algorithms with regard to inaccurate input data.

The key results show that two of our proposed algorithms, namely Greedy-Z and Greedy-C, work best for the two variants, respectively. These algorithms also significantly outperform the well-known set covering algorithm in most cases, especially when interactivity requirement is high.

The rest of the paper is organized as follows. Section II formulates the server provisioning problem and its two variants. The proposed algorithms are presented in Section III. Evaluation methodology is described in Section IV. Experiment results are summarized and analyzed in Sections V. Related work is discussed in Section VI, and Section VII concludes the paper.

II. The interactivity-constrained server provisioning problem

A. System models and assumptions

In this paper, we assume a geographically distributed server architecture with well-provisioned, low-latency inter-server network links (Figure 1). The whole virtual world is partitioned into a number of distinct zones, with each zone managed by only one server.

We further assume that server capacity is unbounded, i.e., a server can take an unlimited number of clients. In fact, a server in this paper should be better referred to as a *server site*, e.g., a data center, or a portion of a data center. We argue that all the costs such as server hardware cost, communication and power infrastructure cost, administration cost, etc. involved in setting up a server site would be significantly higher than simply increasing the capacity of an existing site. Therefore, the focus of this paper is to minimize the number of server sites used without considering capacity of each site. For simplicity, we use the term "server" and "server site" interchangeably from now on.

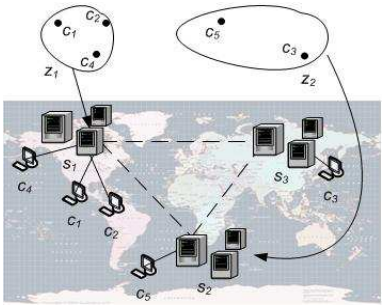


Fig. 1. Distributed server architecture

With the presence of low-latency inter-server links, a client in DVE may connect to its server directly or indirectly. In this paper, we define a client's *contact server* as the server that the client directly connects to. Clients send inputs only to their contact servers. The contact server may execute the input and respond to the client if it is hosting the client's zone , or it may forward the input to another server which is hosting the client's zone. For example, in Figure 1, server s_1 is the contact server of three clients c_1, c_2 and c_4, while c_3's contact server is s_3.

We also define a client's *target server* as the server hosting the client's zone. Inputs from a client will always be forwarded to its target server. The target server may respond to the client directly if it is also the contact server of the client, or it may respond indirectly via the client's contact server. All clients in a zone have the same target server (therefore, we may say "the target server of a zone"), while they may have different contact servers. In Figure 1, we have two zones z_1 and z_2 mapped to servers s_1 and s_2, respectively. Server s_1 is both the contact and target server of c_1, c_2 and c_4, while s_2 is the target server of c_3 and c_5. s_2 is also the contact server of c_5, while c_3's contact servers is s_3. Inputs from c_3 are forwarded to s_2 via s_3.

Similar to previous work in this area, e.g., [5], [9], we assume that the client-server communication latency in DVE is solely determined by the client-server network latency. This is because generally it would be easier to reduce the processing time at the server side by adding more computing power than to reduce message transmission delays in the network.

We also assume that the locations of clients in the virtual world are known in advance, i.e., we know the zone of each client. We further assume that clients would stay in their zones for a considerably long period, e.g., a few months, which means that server re-provisioning will not take place frequently. This is a reasonable assumption, as this is the case with the current well-known MMORPGs in which players spend months exploring a zone, while the game operator keeps releasing content updates which continue to retain the players.

[1]We use various latency models/datasets in this paper as currently there is no single model that can reflect all the properties of Internet latency

B. Problem statement

The following notations are used in the problem statement.

- $C = \{c_1, ..., c_k\}$ - The set that consists of all clients c_i in the DVE.
- $Z = \{z_1, ..., z_n\}$ - The set that consists of all zones i in the DVE, where z_i denotes all clients in zone i.
- $S = \{s_1, ..., s_m\}$ - The set that consists of all servers s_i in the DVE.
- $s^c(c_i)$ - The contact server of a client c_i.
- $s^t(c_i)$ - The target server of a client c_i.
- $d_{c_i s_j}$ - The round-trip network delay between a client c_i and a server s_j.
- $d_{s_i s_j}$ - The round-trip inter-server network delay between server s_i and server s_j.
- D - The *delay bound* of a DVE. The delay bound indicates the maximum round-trip communication delay between a client and its target server to maintain a desired level of interactivity for the DVE. For different types of DVEs, there are different delay bound requirements. For example, the delay bounds of First-Person Shooter (FPS) games and car-racing games are about $250ms$ [10] and $100ms$ [11], respectively. It is also possible for zones in a DVE to have different delay bounds, for instance, each zone hosts a different type of games.

For interactive applications like DVEs, communication delay is the most important *Quality of Service* (QoS) parameter that the system provides to clients [10]. In this paper, we say that a client is *with QoS* or *without QoS* if the communication delay between the client and its target server is less or equal to the delay bound, and otherwise. For instance, we say client $c_k \in z_j$ is with QoS if $d_{c_k s^c(c_k)} + d_{s^c(c_k) s^t(c_k)} \leq D$. Note that if $s^c(c_k) = s^t(c_k)$ then $d_{s^c(c_k) s^t(c_k)} = 0$.

The interactivity-constrained server provisioning problem can be stated as follows. Given the sets C, Z and S of all the clients, zones and servers, find the smallest subset of servers $S_L \in S$ to manage C and Z so that a predefined QoS requirement is satisfied.

The DVE administrator could adjust the QoS parameter to fine-tune the performance objective based on some business strategies. Depending on the definition of the QoS requirement, we have two variants of the server provisioning problem, namely *QoS requirement considering zones* (QoSZ), and *QoS requirement considering clients* (QoSC).

The first variant, QoSZ, requires that each zone in the system is *with QoS*, i.e., satisfying the given QoS requirement $pQoSZ$, as shown in Equation (1). For example, if $pQoSZ = 0.9$, then an optimal solution to this variant should provision the least number of servers so that each zone in the DVE has at least 90% of its clients with QoS.

$$\frac{|\{c_k \in z_j : d_{c_k s^c(c_k)} + d_{s^c(c_k) s^t(c_k)} \leq D\}|}{|z_j|} \geq pQoSZ, \forall z_j \in Z \quad (1)$$

The second variant, QoSC, does not require QoS guarantee for every zone. Instead, it considers the percentage of all clients in the system that are with QoS, as shown in Equation (2).

$$\frac{|\{c_k \in C : d_{c_k s^c(c_k)} + d_{s^c(c_k) s^t(c_k)} \leq D\}|}{|C|} \geq pQoSC \quad (2)$$

One of the reasons why we would need two such different formulations is that, in some scenarios, it is sometimes not possible to find a solution for the QoSZ variant due to the strict QoS requirement, i.e., every zone needs to satisfy the given $pQoSZ$. Secondly, client distributions across zones may be uneven due to time-zone differences and personal preferences. In such cases, each zone should not be treated as equal in terms of QoS consideration, since the number of clients per zone may vary greatly. Thirdly, due to business objectives, it may not be worthwhile to provision many extra servers to satisfy the QoS requirement for all the zones. In these situations, the QoSC variant offers more flexibility to the decision maker.

We also note that the QoS requirement for the QoSZ variant may be harder to satisfy compared to that for the QoSC variant, given $pQoSZ = pQoSC$. This is because in the former variant, every zone needs to meet the given $pQoSZ$, while it is not the case for the latter, i.e., some zones may have better or worse QoS than $pQoSZ$. Therefore, we have the following important remark.

Remark II.1. *Assuming $pQoSZ = pQoSC$, any valid solution for the QoSZ variant will also be a valid solution for the QoSC variant, but not vice versa.*

C. Complexity analysis

Theorem II.1. *The two variants QoSZ and QoSC of the interactivity-constrained server provisioning problem are both NP-hard.*

Proof: We first show that the QoSZ variant is NP-hard. Considering a special case of this problem, in which we assume that the inter-server network is not well-provisioned, hence there is no benefit for each client to relay traffic to its target server from a different contact server due to the large inter-server network latencies. Hence, each client will have the same server as both its target and contact servers. The QoSZ variant now becomes exactly the same as the set covering problem [12], in which a set is defined as the set of zones "covered" by a single server. A server s_i is said to "cover" a number of zones if it satisfies the QoS requirement $pQoSZ$ for each of these zones. Since the QoSZ variant generalizes the set covering problem, which is NP-hard, it is also NP-hard.

The proof for the QoSC variant is similar. We again consider a special case of this variant, in which each client's target and contact servers are the same. We further assume that $pQoSC = 1$, i.e., we need to provide QoS for all clients in the system. This special case of the QoSC variant is also a set covering problem, in which a set is defined in the same manner as in the above proof. ■

III. Server provisioning algorithms

Since both variants of the interactivity-constrained server provisioning problem are NP-hard, in this paper we will focus more on proposing computationally efficient heuristics that

provide decent approximate solutions. In the following, we first describe algorithms for the QoSZ variant. Algorithms for the QoSC variant are actually modified versions of those for the QoSZ variant, hence we will briefly discuss them later in this section.

A. Algorithms for the QoSZ variant

1) Greedy-Z: Algorithm 1 shows the details of the Greedy-Z algorithm. Basically, for each iteration, Greedy-Z considers a new, unselected server s_i. This server is then added into a temporary server set $S'_L = S_L \cup \{s_i\}$, where S_L is the set of already selected servers. The algorithm determines the set of zones with QoS provided by S'_L in line 7 of Algorithm 1. Then, the server s_{max} (among all unselected servers s_i) that results in the largest number of zones with QoS is added into S_L. The algorithm terminates successfully if it can find a set of servers S_L that provides QoS for all the zones, otherwise it returns "NULL".

In line 7 of Algorithm 1, we determine a set of zones with QoS provided by a given set of servers S'_L as follows. For each zone z_j in the system, we check if it is possible to find a server $s_i \in S'_L$ to be z_j's target server, and a set of servers in S'_L to serve as contact servers for all clients in z_j. The goal is to satisfy $pQoSZ$ for z_j, i.e., $\frac{|\{c_k \in z_j : d_{c_k s^c(c_k)} + d_{s^c(c_k) s_i} \leq D\}|}{|z_j|} \geq pQoSZ$, where $s^c(c_k) \in S'L$. Note that a zone has a single target server, while clients in a zone may have different contact servers. If such servers can be found in S'_L, we say that S'_L provides QoS for z_j. We repeat the procedure for all zones in the system to get the total number of zones with QoS provided by S'_L.

Algorithm 1: The Greedy-Z algorithm

Data: sets of servers, zones and clients S, Z, C
Result: set of selected servers S_L, or NULL if no solution is found

1 **begin**
2 initialize the set of zones with QoS $Z^c = \Phi$, $S_L = \Phi$;
3 **while** $|Z^c| < |Z|$ **do**
4 initialize $s_{max} = NULL$;
5 **foreach** $s_i \in S \setminus S_L$ **do**
6 $S'_L = S_L \cup \{s_i\}$;
7 determine the set of zones Z' with QoS provided by S'_L ;
8 if $|Z'| \geq |Z^c|$, assign $s_{max} = s_i$ and $Z^c = Z'$;
9 **end**
10 $S_L = S_L \cup \{s_{max}\}$;
11 if $|S_L| = |S|$ and $|Z^c| < |Z|$, return NULL;
12 **end**
13 return S_L;
14 **end**

Remark III.1. *The complexity of the Greedy-Z algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

Proof: In Algorithm 1, the main loop (line 3-12) will be executed at most m time. The inner loop (line 5-9) of the algorithm determines a set of zones with QoS provided by a set of servers, which requires $O(m^2k)$. This is because for each zone, we need to select a pair of contact and target servers that can provide QoS for that zone among a maximum number of m servers. In addition, to check if a zone is with QoS for a given pair of contact-target server, we need to check all the clients of that zone in the worst case. Hence, the Greedy-Z algorithm requires $O(m^3k)$. ■

The Greedy-Z algorithm shares some similarity with the well-known set covering algorithm [12], in which a server and a zone in Greedy-Z correspond to a set and an element in the set covering algorithm, respectively. The set covering algorithm has a known approximation ratio of $\ln n$, where n is the number of elements that need to be covered.

The key difference between the two algorithms is due to well-provisioned inter-server network links in the QoSZ formulation. Therefore, a server s_i can "cover", i.e., provide QoS for, different sets of zones, depending on other servers in the already selected server set. In the original set covering problem, a set only covers a fixed number of elements, and there is no dependency among the sets.

For example, let's assume we have two servers s_1 and s_2. The QosZ requirement $pQoSZ$ is set to 1, and the delay bound D is set to $100ms$. Each server can provide QoS for one zone (namely z_1 and z_2, respectively), i.e., 2 zones in total if we consider the servers separately. It is possible that the set $\{s_1, s_2\}$ can provide QoS for more than 2 zones, due to the low-latency inter-server links which may reduce client-target server delays. For example, consider another zone z_3 with two clients c_1 and c_2. We further assume that $d_{c_1 s_1} = 50ms$, $d_{c_1 s_2} = 150ms$, $d_{c_2 s_1} = 150ms$, $d_{c_2 s_2} = 50ms$ and $d_{s_1 s_2} = 50ms$. It is clear that either s_1 or s_2 alone will not be able to provide QoS to z_3, given $D = 100ms$ and $pQoSZ = 1$. However, considering the inter-server latency $d_{s_1 s_2} = 50ms$, we may assign z_3 to either s_1 or s_2, and still meet the QoS requirement. For example, if z_3 is assigned to s_2, then $d_{c_1 s_2} = d_{c_1 s_1} + d_{s_1 s_2} = 100ms$.

The above example is not applicable for the original set covering algorithm. For example, the maximum number of elements that a combination of two sets (each covering 1 different element) can cover is 2.

Due to the relationship between Greedy-Z and the set covering algorithm in [12], we have the following remark.

Remark III.2. *Assume a special case of the QoSZ variant in which inter-server network links are not well-provisioned. In this case, the Greedy-Z algorithm provides an approximation ratio of $\ln n$ to the optimal solution, where n is the number of zones.*

We note that finding a tight approximation bound for the general case of the QoSZ variant is a challenging problem of its own, and we leave it for future work.

2) SetCover-Z: The SetCover-Z algorithm implements the server selection strategy originally proposed for the set covering problem [12]. The pseudo-code for the algorithm is shown in Algorithm 2. Such strategy does not consider inter-server dependency, i.e., the merit of each server is assessed individually. In each iteration of Algorithm 2, we find a

Algorithm 2: The SetCover-Z algorithm

Data: sets of servers, zones and clients S, Z, C
Result: set of selected servers S_L, or NULL if no solution is found

1 **begin**
2 initialize the set of zones with QoS $Z^c = \Phi$, $S_L = \Phi$;
3 **while** $|Z^c| < |Z|$ **do**
4 initialize $s_{max} = NULL$, $n\_withQoS = 0$;
5 **foreach** $s_i \in S \setminus S_L$ **do**
6 initialize $Z' = \Phi$;
7 determine the set of zones $Z' \subseteq Z \setminus Z^c$ with QoS provided by only s_i;
8 if $|Z'| \geq n\_withQoS$, assign $s_{max} = s_i$ and $n\_withQoS = |Z'|$;
9 **end**
10 $S_L = S_L \cup \{s_{max}\}$;
11 determine the set of zones Z^c with QoS provided by S_L;
12 if $|S_L| = |S|$ and $|Z^c| < |Z|$, return NULL;
13 **end**
14 return S_L;
15 **end**

single server s_{max} providing QoS for the largest number of *remaining* zones, i.e., those zones that are not already in Z^c. This server is then added into the list of selected servers S_L. The algorithm terminates successfully if it can find a set of servers S_L that provides QoS for all the zones, otherwise it returns "NULL".

Note that line 7 of Algorithm 2 determines the number of zones with QoS provided by each server s_i differently from Greedy-Z. That is, SetCover-Z does not consider any other servers in conjunction with the server s_i in question. This is the same as in the original set covering algorithm which considers each set separately. As a result, a zone with QoS provided by s_i in line 7 of Algorithm 2 would have that same server as both its target and contact servers. However, to be fair when checking the termination condition, line 11 of Algorithm 2 calculates the number of zones with QoS using the entire set of servers that have been selected up to this point. The calculation in this step is similar to that in line 7 of Algorithm 1, i.e., fast inter-server links would be utilized.

Remark III.3. *The complexity of the SetCover-Z algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

Proof: In Algorithm 2, the main loop (line 3-13) will be executed at most m time. The inner loop (line 5-9) of the algorithm determines a set of zones with QoS provided by a single server, which requires $O(mk)$. On the other hand, line 11 of Algorithm 2 requires $O(m^2k)$. Hence, the SetCover-Z algorithm requires $O(m^3k)$. ■

3) Random-Z: The Random-Z algorithm (Algorithm 3) serves as a reference point for comparison against Greedy-Z and SetCover-Z. At each iteration of this algorithm, we add a randomly selected server s_i to the set S_L. We determine the set of zones with QoS provided by all servers in S_L in a similar way to the Greedy-Z algorithm. This algorithm terminates successfully when all zones are with QoS, otherwise it returns "NULL".

Algorithm 3: The Random-Z algorithm

Data: sets of servers, zones and clients S, Z, C
Result: set of selected servers S_L, or NULL if no solution is found

1 **begin**
2 initialize the set of zones with QoS $Z^c = \Phi$, $S_L = \Phi$;
3 **while** $|Z^c| < |Z|$ **do**
4 randomly select a server $s_i \in S \setminus S_L$;
5 $S_L = S_L \cup \{s_i\}$;
6 determine the set of zones Z^c with QoS provided by S_L;
7 if $|S_L| = |S|$ and $|Z^c| < |Z|$, return NULL;
8 **end**
9 return S_L;
10 **end**

Remark III.4. *The complexity of the Random-Z algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

Proof: In Algorithm 3, the main loop (line 3-8) will be executed at most m time. Line 6 of the algorithm determines a set of zones with QoS provided by a set of servers, which requires $O(m^2k)$. Hence, the Random-Z algorithm requires $O(m^3k)$. ■

4) Optimal-Z: The Optimal-Z algorithm (Figure 4) finds the best possible solution for the QoSZ variant. It does so by considering and evaluating all possible combinations of potential servers, and selecting the smallest subset of servers that provide the required level of QoS. We should note that due to the exponential complexity (Remark III.5), this algorithm is only applicable for small instances of the QoSZ variant.

Algorithm 4: The Optimal-Z algorithm

Data: sets of servers, zones and clients S, Z, C
Result: set of selected servers S_L, or NULL if no solution is found

1 **begin**
2 initialize the set of zones with QoS $Z^c = \Phi$, $S_L = S$;
3 find the set $P(S)$ containing all subsets of S;
4 **foreach** $S' \in P(S)$ **do**
5 determine the set of zones Z^c with QoS provided by S';
6 if $|Z^c| = |Z|$ and $|S'| < |S_L|$, $S_L = S'$;
7 if $|S_L| = |S|$ and $|Z^c| < |Z|$, return NULL;
8 **end**
9 return S_L;
10 **end**

Remark III.5. *The complexity of the Optimal-Z algorithm is $O(m^2k2^m)$, where m is the number of servers, and k is the number of clients.*

Proof: In Algorithm 4, the main loop (line 3-8) will be executed 2^m times. Line 5 of the algorithm determines a set of zones with QoS provided by a set of servers, which requires $O(m^2k)$. Hence, the Optimal-Z algorithm requires $O(m^2k2^m)$. ■

B. Algorithms for the QoSC variant

In the QoSC variant, the optimization objective is to minimize the number of servers provisioned to ensure the *percentage of clients with QoS* in the system is at least equal to the given QoS requirement $pQoSC \times 100\%$. As algorithms designed for this variant consider all clients in the system rather than each individual zone; there may be some zones with higher or lower QoS levels than $pQoSC$ in the end. All the three QoSC algorithms in the following discussion are modified versions of the corresponding QoSZ algorithms; hence we will just briefly describe each of them while highlighting the key differences with those for the QoSZ variant.

1) Greedy-C: The Greedy-C algorithm is similar to the Greedy-Z algorithm (Algorithm 1). The main difference between the two is due to the QoS requirement for each variant. In each iteration, Greedy-C selects the server that, if added into the already selected server set, will result in the largest number of *individual clients* with QoS.

The algorithm determines the number of clients with QoS provided by a set of servers S'_L as follows. For a zone z_j in the system, we do a similar check as in Greedy-Z to find a target server $s_k \in S'_L$, and a set of contact servers $S^c \subseteq S'_L$, so that the QoS level of z_j would be *maximized*[2]. This procedure would be repeated for each zone to get the total number of clients with QoS provided by S'_L. The algorithm terminates if the percentage of clients with QoS in the system is equal to or larger than $pQoSC \times 100\%$, or all servers have been used. Note that there may be remaining zones that have not been checked when the algorithm terminates. These zones can be assigned to any of the selected servers later, as their QoS levels do not contribute towards satisfying the given $pQoSC$ requirement.

Remark III.6. *The complexity of the Greedy-C algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

Proof: The proof is similar to that for Remark III.1. ■

2) SetCover-C: The SetCover-C follows a similar server selection strategy outlined in Algorithm 2. At each iteration, SetCover-C selects the server s_{max} that provides QoS for the largest number of remaining *individual clients*. This algorithm would terminate when the percentage of clients with QoS in the system becomes equal to or larger than $pQoSC \times 100\%$, or all servers have been used.

[2]The best case is that all clients in z_j are with QoS.

Remark III.7. *The complexity of the SetCover-C algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

3) Random-C: At each iteration of this algorithm, a randomly selected server s_i will be added into the set of already selected servers S_L. We determine the set of clients with QoS provided by S_L in a similar way to Greedy-C. The terminating condition for Random-C is the same as those for Greedy-C and SetCover-C. The pseudo-code for this algorithm is similar to Algorithm 3.

Remark III.8. *The complexity of the Random-C algorithm is $O(m^3k)$, where k is the number of clients and m is the number of servers.*

4) Optimal-C: This algorithm is similar to Optimal-Z, except that we determine the set of clients with QoS provided by a set of servers similarly to Greedy-C. Optimal-C is also applicable to small instances of the QoSC variant only.

Remark III.9. *The complexity of the Optimal-C algorithm is $O(m^2k2^m)$, where k is the number of clients and m is the number of servers.*

IV. Evaluation methodology

For more realistic and reliable algorithm evaluation, we employ both the traditional approach of using static client-server round-trip latency data [4], [9]; as well as dynamic latency data obtained via our own real-world deployment of a DVE interactivity monitoring system. Considering the high overhead of direct latency measurement in reality, we also conduct experiments with iPlane [8], an Internet latency prediction system. We use latency estimations from iPlane to assess the robustness of the proposed provisioning algorithms in the presence of inaccurate input data.

Since the workload distributions in real-life DVEs are hardly uniform, we also consider various client distributions in both the network as well as in the virtual world; and the correlation between each client's network location and its virtual world location.

A. Static latency data

For the static latency model, we largely follow the methodology presented in [4]. To generate realistic round-trip latency data between all servers and clients in the system, we employ the tool DS^2 (Delay Space Synthesizer) [13]. DS^2 takes real latency measurements[3] as inputs and builds models that exhibit most of the essential properties of the Internet latency space, such as overall delay distribution, global and local clustering, delay growth metrics, triangle inequality violations, etc. The first two rows of Table I list the DS^2-generated network latency datasets used in this paper. Each has a maximum round-trip latency of 1000 ms.

[3]Real-life, large-scale Internet latency measurements often produce incomplete data due to various reasons, e.g., network/server outage. DS^2 can interpolate missing measurements, which is very convenient for our simulation.

TABLE I
STATIC LATENCY DATA

Name	Model	Nodes	Links
REAL1	DS^2, Matrix1 from [6]	3000	-
REAL2	DS^2, Matrix2 from [6]	3000	-
BRITE1	Waxman	3000	6000
BRITE2	Barabasi-Albert	3000	5997

For diversity, we also employ latency data based on network topologies generated by the popular topology generator BRITE [7]. The last two rows of Table I list those datasets. End-to-end network latencies for each topology are calculated following shortest-path routing. The link latency values in these topologies are generated by BRITE, which are proportional to the physical distance between the two nodes of each link. Each topology has a maximum round-trip latency of 300 ms.

B. Dynamic latency data

Previous studies in DVE interactivity enhancement, for example [4], [5], [9], assume static pair-wise client-server round-trip Internet latencies. In reality, Internet latency fluctuates frequently due to unexpected network load, routing problem, router/server failures, etc. Any networking algorithm, not just those designed specifically to improve DVE interactivity, that relies on measured latency or bandwidth at one time might not be working well at a later time. Therefore, it is desirable to examine the performance of the proposed server provisioning algorithms considering such realistic Internet conditions. On the other hand, the large-scale, distributed and real-time nature of typical DVEs makes it even more challenging to collect sufficient and reliable data to judge whether a newly developed approach does indeed produce tangible performance improvement.

For this purpose, we have proposed and developed a scalable and extensible software framework named DINE (DVE INteractivity Evaluation) [14] to support the development, integration, and performance evaluation of network latency based methods for improving the interactive performance of large-scale DVEs [14]. The framework should be flexible enough to serve as either an evaluation platform for the development of new DVE interactivity enhancement methods, as well as a real-world performance monitoring and management suite for existing DVEs. In this paper, we use DINE to examine the performance of the server provisioning algorithms under dynamic Internet condition. For ease of reference, key features of DINE are briefly mentioned below.

1) The DINE framework: The core of DINE is an efficient tool to measure, analyze and visualize DVE interactive performance. Some of the most important design goals of the DINE tool are scalability, fault-tolerance and extensibility. First, the tool needs to be scalable, as DVE interactivity measurement might involve thousands of distributed clients interacting concurrently in the virtual world. Second, as mentioned previously, the Internet may be unreliable at times, as server and network outages are common problems. The tool must have the capability to handle such unexpected events to avoid erroneous measurements which may lead to misleading

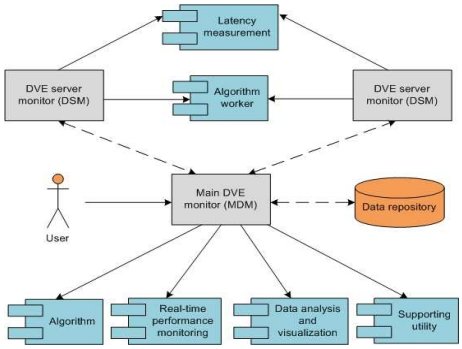
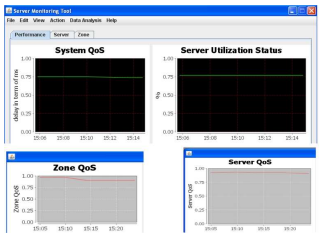
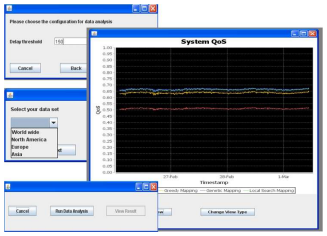


Fig. 2. The DINE tool's components and extensible modules



(a) Real-time monitoring



(b) Data analysis and visualization

Fig. 3. Screenshots of the DINE tool in action

conclusions. Third, as new, more efficient networking tools continue to emerge, our tool needs to be able to take advantage of such developments easily.

Figure 2 shows the tool's key software components and extensible modules used by each component. In particular, the two main components are the "main DVE monitor" (MDM) and the "DVE server monitor" (DSM). The tool and its components have been implemented in Java for better portability. Figure 3 shows some screenshots of the tool.

- **The MDM component:** The MDM runs on a central server and has a GUI-based interface for ease of use by DVE administrators or algorithm developers. It provides functionalities such as simulation suppporting tools, algorithm integration, real-time performance monitoring, measurement error handling, data post-processing, analysis and visualization. These functionalities are designed as "plugins", i.e., developers/researchers can develop their own modules with enhanced functionality as they wish; and "plug" them into the DINE architecture in a straightforward manner.

• **The DSM component** The DSM is a light-weight component running on each DVE server. In our prototype implementation, it has two main tasks. The first task involves receiving and applying algorithm execution's outcome sent from the MDM's Algorithm module. The second one collects performance information regarding each server, and sends back to the MDM for consolidation.

Various performance indicators can be collected, e.g., CPU and memory utilization, network load, server downtime, etc. In this paper, we focus on collecting round-trip client-server network latency using direct measurement and prediction. It is well-known that large-scale data collection on the Internet, with thousands of clients' and servers' locations, is a very challenging task on it own [13]. Latency prediction/estimation techniques, such as Meridian [15] or iPlane [8] may have an advantage due to their scalability, however the accuracy could be compromised. On the other hand, direct measurement using the simple "ping" command, or indirect method like King [16] are likely to provide more accurate results at the cost of higher measurement overhead. For this paper, we have implemented a scalable direct measurement mechanism using"ping" in the DINE tool. This is the main method to collect latency data periodically over time for the performance evaluation in this paper.

Due to the high overhead of direct measurements, we also employ latency prediction using iPlane [8], which is a scalable service developed at the University of Washington for predicting Internet path performance[4]. The service constructs a structural model of the Internet, and predicts end-to-end performance by composing measured performance of segments of known Internet paths. iPlane can predict Internet latency, bandwidth, loss rates, etc. among other things between arbitrary Internet hosts.

In addition to real, directly measured data, we use iPlane's latency predictions to construct the input data for the server provisioning algorithms. Then, we continuously monitor the performance variation of the algorithms over time using real, direct measurements. The primary goal of such experiments is to assess the robustness and resiliency of the proposed algorithms under real Internet conditions, where accurate input data are hard and expensive to collect.

2) Deployment of the DINE framework: To evaluate the proposed server provisioning algorithms with dynamic latency data, the DINE tool is deployed over PlanetLab (http://www.planet-lab.org). PlanetLab is an Internet-scale research network testbed that supports the development of new network services. Many researchers from both academia and industry have used this platform to develop and evaluate new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing. At the time of writing, PlanetLab has 1074 nodes at 496 sites distributed all over the Internet.

For latency measurement/predictions, we collect the sets of servers' network locations (IP addresses) and clients' network locations from PlanetLab. The 28 potential DVE servers are assumed to be physically located at the 28 network locations listed in Table II. These servers are also assumed to have unlimited capacity. At each server location, we deploy an instance of the DSM. The MDM runs on a dedicated server in our institution's network. The communication between instances of the DSM and the MDM is done via TCP.

TABLE II
LOCATION DISTRIBUTION OF SERVERS ON PLANETLAB

Region	North America	Europe	Asia	Others
Num. of locations	7	10	7	4

The dynamic latency data collection for this paper started on Jun 24, 2010 2:27:16 AM and ended on Jul 7, 2010 1:04:26 PM, Singapore local time. There are originally over 800 clients' network locations used for the data collection. After filtering out unresponsive ones that caused measurement errors in both DINE and iPlane, we are left with 497 locations (Table III). Each simulated client plays in one of the virtual world zone, and is assumed to be physically located at one of the 497 network locations. The number of clients per location may vary according to the chosen workload distribution discussed below. We note that there are more locations available from North America and Europe compared to other regions. This is due to the degree of contribution/participation into PlanetLab of countries in each continent.

With 28 servers and 497 client locations, we still have missing and erroneous measurements between some client-server pairs in our dynamic dataset. We use DINE's utilities to pre-process the dataset as follows. For any pair without a valid measurement, or with a very high value (e.g., more than 5 seconds), we will give it the median of all the corresponding values collected by the end of the measurement. We have high confidence in the final dataset since the total number of such replacements constitute less than 0.8% of the entire data.

TABLE III
LOCATION DISTRIBUTION OF CLIENTS ON PLANETLAB

Region	North America	Europe	Asia	Others
Num. of locations	216	197	57	27

C. Workload models

TABLE IV
DISTRIBUTION TYPES

Type	**0**	**1**	**2**	**3**
Client cluster in VW	No	Yes	No	Yes
Client cluster in NW	No	No	Yes	Yes

1) Client distributions: For more robust and reliable simulation results, we have simulated a number of combinations of client distributions in both the virtual world (VW) and the network (NW). Table IV shows these combinations. The rationale for simulating various distributions is that the number of clients may be larger in some specific zones of the virtual world than others. This is due to the clustering of clients

[4]We have also tested out freely available prediction systems such Meridian [15] and Pyxida [17]. However, Meridian does not provide latency estimation between arbitrary hosts, while Pyxida requires significant deployment efforts for the scale of our measurement (more than 800 hosts on PlanetLab).

in these more popular zones, e.g., those with more game resources would attract more players. In the network, due to the differences in time zones of geographically distributed clients, at a specific time, the number of online clients in the DVE may be quite different for different geographic regions [18].

To simulate the clustering behavior of clients in the virtual world or the network, we randomly select some zones or network locations to have more clients than other zones/locations. More specifically, in the clustered distribution in the virtual world, a popular zone would have about 4 times more clients compared to an average zone. For the network, clustered locations would have about 3 times more clients than the rest. Similar performance trends have been observed throughout the study with varying cluster sizes and numbers.

2) Correlation factor: To model the relationship between clients' locations in the network and in the virtual world, we use a correlation factor denoted as δ, where $0 \leq \delta \leq 1$ [19]. The higher the value of δ, the stronger the tendency for clients from the same network location to gather in the same zone of the virtual world. The rationale for this correlation factor is that, in general, we should note that clients gathering in the same zone of a DVE may not necessarily be close to each other in terms of their network locations. On the other hand, it is natural to observe that clients that are close to each other in their physical locations (e.g., from the same country or the same geographic region) tend to gather in a specific zone of the virtual world since they may share similar cultural preferences. The correlation between these two kinds of client locations may have some impacts on the performance of our server provisioning algorithms.

D. Default settings and parameters

Unless otherwise stated, the following assumptions and default settings are used in the experiments. The clients are uniformly distributed in the network as well as in the virtual world, and the correlation factor is zero. There are 5000 clients coming from 100 uniformly selected network locations, 100 zones, and 100 servers, each having unlimited capacity. Server locations are uniformly distributed in the network. The DVE delay bound D is set to $100ms$ to simulate high interactivity requirement. We also conduct experiments with the delay bounds of zones ranging from $50ms$ to $150ms$. The default QoS requirements are $pQoSZ = 0.8$ and $pQoSC = 0.95$. The default latency dataset is REAL1. Similar to [20], the latencies on inter-server network links are set to 10% of the original values directly obtained from the latency datasets to emulate the well-provisioned inter-server network links.

V. Results and analysis

In this section, we describe the experiment results which are the average of 50 independent simulation runs.

A. Performance of QoSZ algorithms

1) Static latency data: We first present the experiment results for the static latency datasets listed in Table I. Figure 4(a) to 4(c) show the performance of the proposed QoSZ algorithms in terms of number of provisioned servers with various QoS requirements ranging from 0.7 to 0.95. In particular, Figure 4(a) shows the performance of all four QoSZ algorithms with a dataset of 10 potential servers[5]. On the other hand, Figure 4(b) and 4(c) both show the performance of Greedy-Z, SetCover-Z and Random-Z for a larger set of potential servers (100 servers). In Figure 4(c), we vary the delay bounds of zones to reflect possible real-world scenarios, in which each zone may have a different interactivity requirement. More specifically, we assign a randomly selected delay bound of either $50ms$, $100ms$ or $150ms$ to each zone in this experiment.

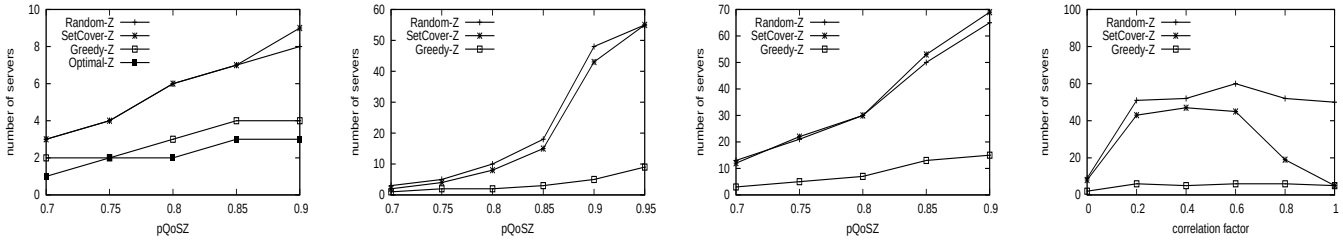
Unsurprisingly, increasing QoS requirement requires more servers to be provisioned in most cases. The figures also show that most of the time Greedy-Z significantly outperforms the other two algorithms (SetCover-Z and Random-Z). More specifically, in Figure 4(b), Greedy-Z outperforms Random-Z by a factor ranging from 2.5 to 9.6, and outperforms SetCover-Z by a factor ranging from 2 to 8.6. The performance improvement of Greedy-Z is much more pronounced as the QoS requirement increases. This illustrates the effectiveness and robustness of Greedy-Z over SetCover-Z and Random-Z. In addition, for the case of small potential server set (Figure 4(a)), Greedy-Z has comparable performance to the optimal algorithm (Optimal-Z) with much less execution time (a few tens of milliseconds compared to several minutes).

Although the selection approach used in SetCover-Z has been working well for traditional set covering problems [12], providing one of the best approximation ratios of $\ln n$ for such problems, it performs far worse when applied here. Figure 4(b) shows that SetCover-Z's performance is not much better compared to a random selection strategy (Random-Z). More specifically, in this experiment, SetCover-Z outperforms Random-Z by a factor ranging from 1.11 to 1.5; and they have similar performance when $pQoSZ = 0.95$.

Such performance behavior is largely due to the fact that inter-server dependency has not been considered when selecting the best server in each of SetCover-Z's iteration. Indeed, a closer examination of SetCover-Z's execution trace reveals that most of the times it can only choose several good servers in the first few iterations. The rest of the selected servers have been chosen rather randomly. This is because by itself, each server selected in the later iterations frequently does not provide QoS to any of the remaining zones. That is, in Algorithm 2, for each s_i considered in later iterations, $n\_withQoS$ is always equal to zero. This also explains why SetCover-Z performs much worse compared to Greedy-Z as $pQoSZ$ increases.

The effect of various correlation factors is shown in Figure 4(d). We note that Greedy-Z needs several additional servers to meet the given QoS requirement when there is a certain degree of correlation, e.g., $\delta \geq 0.2$ in this figure. Recall that δ represents the likelihood of clients coming from the same network location to gather in the same zone of the virtual world. On the other hand, SetCover-Z performs much worse

[5]Recall that the fourth algorithm, Optimal-Z, is only applicable for a small set of potential servers.



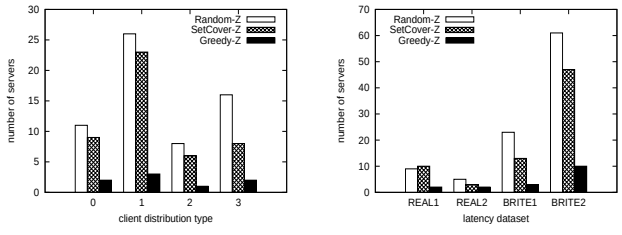
(a) 10 potential servers (b) 100 potential servers (c) 100 potential servers, varied delay bounds (d) Different correlation factors

Fig. 4. Impacts of different QoS requirements and correlation factors - QoSZ variant

as δ increases, which is similar to Random-Z. However, it begins to show improvement as this factor reaches a certain high value. When δ is very high (near 1), SetCover-Z performs similarly to Greedy-Z.

The above observation could be roughly explained as follows. Greedy-Z tends to select servers that are near the central region of the set of clients' network locations. In this experiment, clients are uniformly distributed in the network as well as in the virtual world. If δ is low, i.e., clients in each zone would be coming from many different network locations. In this case, a small number of "centrally" provisioned servers would be sufficient to satisfy the given QoS requirement.

On the other hand, if the correlation factor is very high, e.g., $\delta \approx 1$, most clients coming from the same network location will gather in the same zone. In such case, the fast inter-server network links might become unnecessary. For example, let's assume a DVE with two zones z_1 and z_2. Further assume that all clients coming from Singapore gather in z_1, and all those from New York gather in z_2. In this case, to meet a relatively high QoS requirement, we may need two servers, each located in or near to Singapore or New York, to host zone z_1 and z_2, respectively. There is no need for a fast network link between these two servers. This explains why SetCover-Z, which does not use inter-server dependency in making provisioning decisions, might perform similarly to Greedy-Z when $\delta \approx 1$.



(a) Client distributions (b) Latency datasets

Fig. 5. Impacts of different client distributions and latency datasets - QoSZ variant

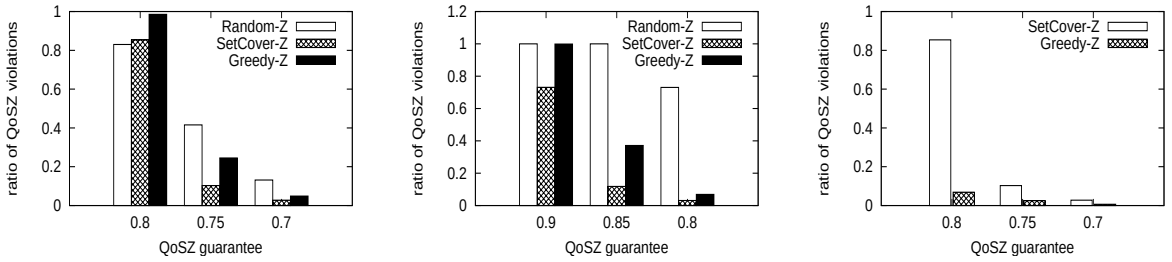
To further verify the performance outcome, we have also tested the proposed algorithms with different client distributions in the network as well as in the virtual world (these distributions are listed in Table IV); and with various latency datasets. The results are summarized in Figure 5(a) and 5(b) respectively. We have observed that when clients are clustered in the network (e.g., distribution type 2), the performance of all algorithms seems to be better compared to the case of no clustering. On the other hand, if the clients are clustered in the virtual world (e.g., distribution type 1), all algorithms perform worse than the case of no clustering. It is also obvious that Greedy-Z outperforms the rest in all clients distributions and datasets with significantly large margins.

2) Dynamic latency data: In this set of experiments, we evaluate the performance of QoSZ algorithms under dynamic Internet conditions. One of the main input data for the algorithms is the pair-wise client-server latency matrix. In order to obtain more reliable results, we use the median values of the first 20 measurements to produce the input latency matrix. The performance behaviors of each algorithm are then observed with the rest of the latency data collected. In this way, we will be able to see the effect of Internet latency variations over time on the server provisioning algorithms.

In this paper, we are interested in the *ratio of QoS violations* of each server provisioning algorithm over time. More specifically, with the QoSZ variant, we determine the percentage of zones with QoS for each round of latency measurement. If the percentage is below 100%, it will be considered as a QoSZ violation. We then compute the ratio of QoSZ violations by dividing the number of times when there is a QoSZ violation to the total number of measurement rounds made.

For the dynamic latency dataset, we use a delay bound $D = 150ms$, 28 potential server locations, and 10000 clients coming from 497 network locations distributed over PlanetLab. Figure 6 shows the ratio of QoSZ violations over our measurement period (around 2 weeks) for all QoSZ algorithms. In this figure, the x-axis shows the possible QoSZ guarantees, while the y-axis shows the corresponding ratio of QoSZ violations for each QoSZ guarantee. For example, Figure 6(a) shows three possible QoSZ guarantees (0.8, 0.75 and 0.7) for the same server provisioning decision made by running the QoSZ algorithms with the input parameter $pQoSZ = 0.8$.

Due to fluctuations in Internet latency, it is anticipated that 100% percent of zones with $pQoSZ = 0.8$ is not sustainable over time. Indeed, Figure 6(a) shows that the ratio



(a) pQoSZ = 0.8

(b) pQoSZ = 0.9

(c) pQoSZ = 0.8 for SetCover-Z, pQoSZ = 0.9 for Greedy-Z

Fig. 6. Impacts of dynamic latency data - QoSZ variant

of QoSZ violations is very high for all three algorithms, if we consider the original QoSZ guarantee of 0.8. However, if we lower the QoSZ guarantee to 0.75 or 0.7, SetCover-Z and Greedy-Z improve greatly in terms of QoSZ violations. More specifically, when we set the lowest QoSZ guarantee to 0.7, the ratios of QoSZ violations for SetCover-Z and Greedy-Z are 0.02 and 0.04, respectively. Similar results have also been observed for different $pQoSZ$, for example $pQoSZ = 0.9$ (Figure 6(b)).

The above observations suggest that, due to varied Internet latency, it is very hard to maintain the original QoSZ guarantee with low ratio of QoSZ violations. Hence, to achieve a certain level of QoSZ guarantee, we may have to over-provision. For example, to get a QoSZ guarantee of 0.8 with low ratio of violations, we may run the provisioning algorithm with $pQoSZ = 0.9$. Figure 6(c) illustrates such scenario. In this experiment, we run Greedy-Z with $pQoSZ = 0.9$ and compare the result against SetCover-Z with $pQoSZ = 0.8$. It is observed that in this case, Greedy-Z can provide a QoSZ guarantee of 0.8 with a very low ratio of violations (around 0.06, compared to 0.85 produced by SetCover-Z at the same QoSZ guarantee). Table V also shows that even with $pQoSZ = 0.9$, Greedy-Z still uses less servers (3) compared to SetCover-Z with $pQoSZ = 0.8$ (5 servers).

TABLE V
NUMBER OF SERVERS SELECTED - QOSZ VARIANT

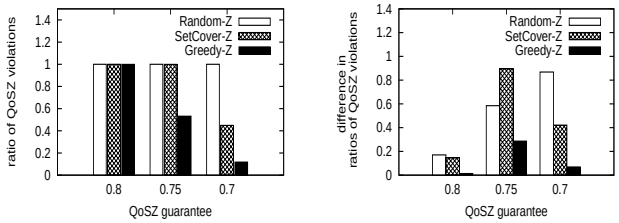
Algorithm	Random-Z	SetCover-Z	Greedy-Z
Figure 6(a)	6	5	2
Figure 6(b)	6	7	3
Figure 6(c)	-	5	3
Figure 7(a)	4	5	2

Next, we evaluate the sensitivity of the QoSZ algorithms with regard to inaccurate input data, which are common in real-world scenarios. As mentioned previously, one of the most important input data for the proposed algorithms is the pair-wise latency matrix between all possible server and client locations. iPlane [8] is a more scalable and less costly alternative compared to direct Internet latency measurements for obtaining such latency matrix. However, the accuracy of the predicted latency by iPlane may be compromised.

We use iPlane to estimate the round-trip latency between any pair of our selected PlanetLab's network locations 20 times, and use the median of those estimations to construct the input latency matrix for the QoSZ algorithms. We refer to this latency matrix as the "estimated input". This is to distinguish from the input latency matrix obtained via direct measurements, which we refer to as the "real input". Table VI shows the estimation errors of the estimated input when comparing against the real input. For example, we can see that 63% of the estimated input having an estimation error less than 20ms. We then use the directly measured, 2-week latency dataset above to evaluate the ratio of QoSZ violations for each algorithm.

TABLE VI
ERRORS IN LATENCY ESTIMATIONS USING IPLANE

$<$ 10ms	$<$ 20ms	$<$ 50ms	$<$ 100ms	$<$ 200ms	$<$ 500ms
45%	63%	81%	90%	97%	98%

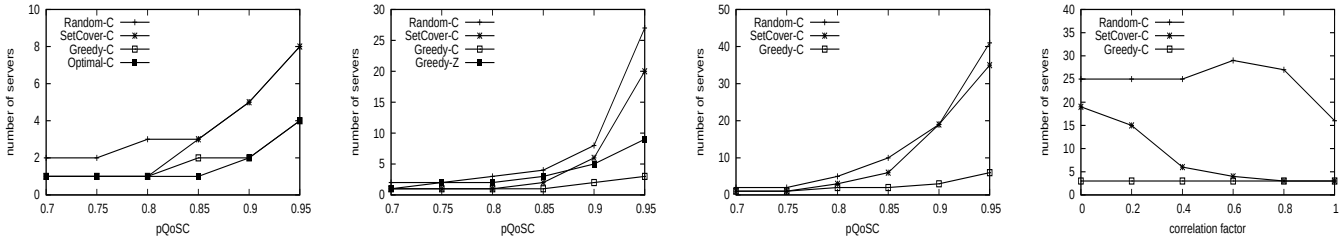


(a) Ratios of QoSZ violations using inaccurate input data obtained from iPlane

(b) Differences in the ratios of QoSZ violations between iPlane and directly measured input data

Fig. 7. Impacts of inaccurate input data obtained from iPlane - QoSZ variant, pQoSZ = 0.8

Figure 7(a) shows that the violation ratio of Greedy-Z is the lowest among the three algorithms. In addition, Figure 7(b) shows the *differences* in the QoSZ violation ratios for each algorithm. We calculate the difference for an algorithm by subtracting its QoSZ violation ratio obtained by using the estimated input to that obtained by using the real input. Such difference reflects the sensitivity of the algorithm to its input data. Figure 7(b) shows that Greedy-Z is less sensitive to inaccurate input data among all algorithms.



(a) 10 potential servers (b) 100 potential servers (c) 100 potential servers, varied delay bounds (d) Different correlation factors

Fig. 8. Impacts of different QoS requirements and correlation factors - QoSC variant

B. Performance of QoSC algorithms

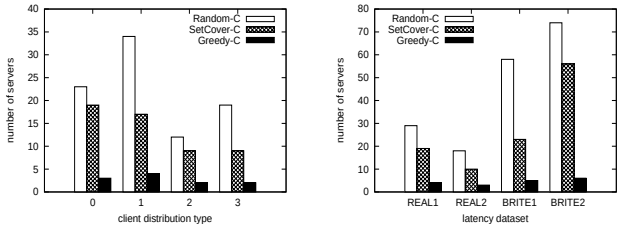
1) Static latency data: Figure 8(a) to 8(c) show the performance results of various QoSC algorithms, plus Greedy-Z (Figure 8(b)). Recall that any algorithms for the QoSZ variant can be used to solve the QoSC variant, but not vice versa. Here we also want to compare the best QoSZ algorithm against those designed specifically for the QoSC variant. From these figures, it is observed that Greedy-C performs comparably to Optimal-C, while outperforms all other algorithms, including Greedy-Z. The latter performs better than SetCover-C when $pQoSC$ is high (Figure 8(b)).

The data obtained from this set of experiments also highlight that the QoS requirement in the QoSZ formulation is harder to meet compared to that in the QoSC formulation. This is illustrated by the smaller number of servers required to meet $pQoSC$, compared to that for $pQoSZ$, especially for higher QoS requirements. For example, in Figure 8(b), using the best algorithm for each problem variant, $pQoSC = 0.95$ requires only 3 servers, whilst the same $pQoSZ$ needs thrice that number.

Figure 8(d) shows the effect of various correlation values on all QoSC algorithms. While SetCover-C's performance improves when δ increases, Greedy-C seems insensitive to this parameter. This is due to the fact that in the QoSC formulation, we count only the number of *individual* clients with QoS, without considering whether the zones that those clients are in would meet the QoS requirement or not. Therefore, the clients' virtual locations are not very important to Greedy-C. On the other hand, SetCover-C performs similarly to Greedy-C when δ becomes high enough, e.g., larger than 0.6 in this experiment. The reason is similar to what we have explained previously for SetCover-Z.

Figure 9(a) and 9(b) show the performance of all QoSC algorithms in different client distributions and latency datasets, respectively. The key observations are similar to those in the previous section, which further confirm the effectiveness of Greedy-C.

2) Dynamic latency data: We now evaluate the performance of QoSC algorithms under dynamic Internet conditions. We use a similar approach to that used for evaluating the QoSZ algorithms. The only difference here is how we calculate the *ratio of QoSC violations*. With the QoSC variant, we determine the ratio of clients with QoS for each round of



(a) Client distributions (b) Latency datasets

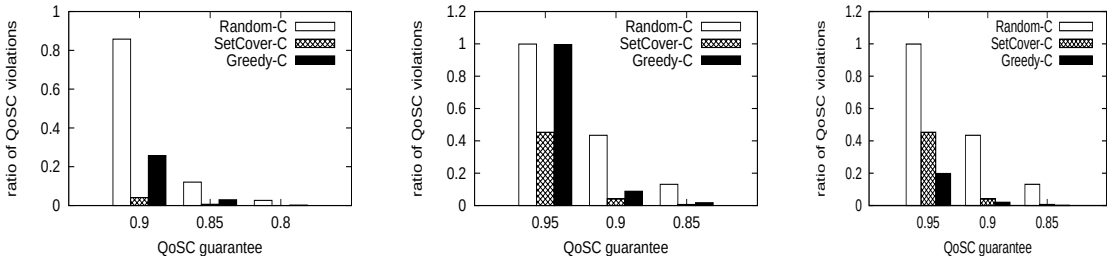
Fig. 9. Impacts of different client distributions and latency datasets

latency measurement. If the ratio is below the given $pQoSC$, then it will be considered as a QoSC violation. The ratio of QoSC violations over the entire measurement period will then be computed similarly to that of the QoSZ variant.

Figure 10 plots the ratio of QoSC violations, while Figure 11 plots the QoSC variations over time of the three propose algorithms. We observe similar effects of latency variation on the QoSC guarantees, as in the previous experiments for the QoSZ variant. More specifically, slightly lowering the original $pQoSC$ requirement reduces the ratio of QoSC violations significantly for all QoSC algorithms. Hence, to meet a certain QoSC requirement with low violation ratio, a DVE infrastructure service provider may choose to carry out over provisioning. For example, to provide a QoSC guarantee of 0.8 in 99% of the time, the service provide may run Greedy-C with an input $pQoSC = 0.9$ (the actual QoSC violation ratio in this case is around 0.002 as shown in Figure 10(a)).

Figure 10(c), 11(b) and the third row of Table VII show another example of over provisioning. Given $pQoSC = 0.95$ and the delay bound $D = 150ms$. We run Greedy-C with a lower delay bound of $125ms$, while SetCover-C is run with the original delay bound of $150ms$. It is noted that Greedy-C now can provide much lower ratios of QoSC violations for all QoSC guarantees but still use less servers compared to the other two algorithms.

Finally, Figure 12(a) shows that the violation ratio of Greedy-C is the lowest among the three algorithms, when using iPlane's estimates as the input latency matrix. Figure 12(b) also shows that Greedy-C is less sensitive to inaccurate input

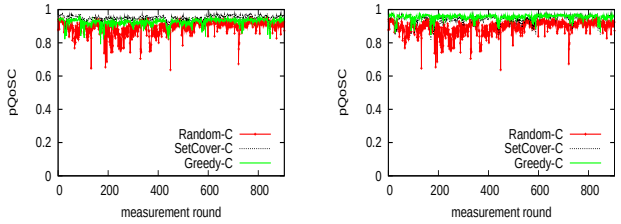


(a) pQoSC = 0.9

(b) pQoSC = 0.95

(c) pQoSC = 0.95, delay bound $D = 150ms$ for SetCover-C and Random-C, $D = 125ms$ for Greedy-C

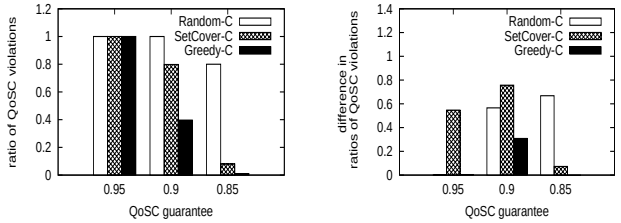
Fig. 10. Impacts of dynamic latency data - QoSC variant



(a) $D = 150ms$ for all algorithms

(b) $D = 150ms$ for SetCover-C and Random-C, $D = 125ms$ for Greedy-C

Fig. 11. Impacts of dynamic latency data - QoSC variant, pQoSC = 0.95



(a) Ratios of QoSC violations using inaccurate input data obtained from iPlane

(b) Differences in the ratios of QoSC violations between iPlane and directly measured input data

Fig. 12. Impacts of inaccurate input data obtained from iPlane - QoSC variant, pQoSC = 0.95

data compared to other QoSC algorithms. This observation is similar to what we have seen in the QoSZ variant.

VI. Related work

Existing research involving server and network infrastructure to improve interactivity for DVEs has usually been formulated as a *resource-oriented* load balancing problem, i.e., the objective is to minimize server processing delay by balancing the workload among servers rather than to reduce the network latency between clients and their servers [21], [22], [23]. Recent work such as [4], [5] had been focusing on efficient client-to-server assignment, referred to as the *zone mapping* approach, to minimize network latency, thus improving interactivity.

TABLE VII
Number of servers selected - QoSC variant

Algorithm	Random-C	SetCover-C	Greedy-C
Figure 10(a)	4	8	2
Figure 10(b) and 11(a)	8	8	3
Figure 10(c) and 11(b)	8	8	4
Figure 12(a)	7	11	3

The zone mapping approach assumes a fixed, capacitated server infrastructure already in place; for which the overall interactivity needs to be maximized given the resource limitation. Our work in this paper, with two novel problem formulations, focuses on a different problem and objective, but could be used to complement the zone mapping approach. For instance, the DVE administrator may use the proposed server provisioning algorithms to provision enough servers to meet the minimum QoS requirement based on his/her business objective and strategies. After that, zone mapping algorithms can be used to further improve the QoS level if needed, or to deal with dynamic changes in the DVE like client joining/leaving/moving around the virtual world.

In [3], several server provisioning algorithms had been proposed to select appropriate locations to place DVE servers. However, [3] does not take into account the virtual locations of clients, thus the improvement over random provisioning is not very significant. Furthermore, in addition to pair-wise client-server network latencies, the provisioning algorithms in [3] also need complete and accurate knowledge regarding underlying AS-level Internet topology, which is quite costly (in terms of resource and time) to obtain reliably.

In another closely related work, an algorithm was proposed in [20] for game clients to select the best server in terms of interactivity in a distributed manner. For the distributed algorithm to work, a mirrored architecture was assumed, which replicates the DVE zones at multiple servers spread across the Internet. This approach shares some similarities

with the web server replica placement problem in Content Distribution Networks (CDNs) [24], [25]. However, unlike web replications, much more complicated consistency issues should be dealt with in DVEs [26]. In either our QoSZ or QoSC formulation, only one server has the control over the state of a zone, thus consistency can be maintained more easily.

It is also noted that our problem formulation shares some similarity with the well-known facility location problem in operations research [27]. This problem can be briefly defined as follows. Given a set of locations i, building a facility at each location i costs f_i. Each client j needs to be assigned to a facility. The cost of such assignment is $d_j c_{ij}$, where d_j and c_{ij} denote the demand of client j and distance between facility i and client j, respectively. The objective is to find the number and location of each facility with the minimum total cost. In our problem, we also aim to minimize the total number of "facilities", i.e., servers used. The main differences are that we need to guarantee a certain level of QoS, and there are inter-server network links which enable a "facility" to indirectly serve clients.

VII. Conclusions

In this paper, we consider a new problem, referred to as the interactivity-constrained server provisioning problem. The main goal is to the minimize resource needed, i.e., the number of servers that need to be provisioned, to achieve a pre-specified QoS requirement in large-scale, highly interactive DVEs. To this end, we have proposed two different formulations for the problem, namely QoSZ and QoSC, and shown that both are NP-hard. These two formulations would offer DVE administrators more flexibility in selecting the right QoS requirements for their DVEs, considering various business constraints they may face in the real world.

A number of computationally efficient heuristics have also been developed for the problem. Extensive simulation study on realistic network models and dynamic latency data collected from large-scale Internet latency measurements have shown that two new greedy algorithms, taking into account inter-server dependency, work best for the QoSZ and QoSC variants, respectively. The experiments with dynamic Internet latency data also suggest some important considerations, e.g., over provisioning sometimes is necessary for real-world deployments of the provisioning algorithms, in light of fluctuations in Internet performance. Last but not least, the best algorithms, namely Greedy-Z and Greedy-C, appear to be quite resilient to inaccurate input data such as the round-trip latency obtained from iPlane. This further confirms the practicality of these algorithms for real-world DVE server provisioning.

Acknowledgement

This work is supported in part by the Singapore National Research Foundation under Grant NRF2007IDM-IDM002-052. The authors would also like to thank Harsha V. Madhyastha, the creator of iPlane [8], for helping to obtain the iPlane's latency estimations.

References

[1] S. Singhal and M. Zyda, *Networked virtual environments: design and implementation*. Reading, MA: Addison-Wesley, 1999.

[2] V. Nae, A. Iosup, S. Podlipnig, R. Prodan, D. Epema, and T. Fahringer, "Efficient management of data center resources for massively multiplayer online games," in *Proc. of ACM/IEEE Supercomputing*, 2008, pp. 1–12.

[3] D. Ta, S. Zhou, R. Ayani, W. Cai, and X. Tang, "Network-aware server placement for highly interactive distributed virtual environments," in *Proc. of IEEE DS-RT*, 2008, pp. 95–102.

[4] D. Ta, S. Zhou, X. Tang, W. Cai, and R. Ayani, "Efficient zone mapping algorithms for distributed virtual environments," in *Proc. of ACM/IEEE/SCS PADS*, 2009, pp. 137–144.

[5] D. Ta and S. Zhou, "A two-phase approach to interactivity enhancement for large-scale distributed virtual environments," *Computer Networks*, vol. 51, no. 14, pp. 4131–4152, 2007.

[6] "Internet delay space synthesizer - datasets," *Available at http://www.cs.rice.edu/ bozhang/ds2/matrix/*, retrieved Jul 2010.

[7] "Brite internet topology generator," *Available at http://www.cs.bu.edu/brite*, retrieved Jul 2010.

[8] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. E. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," in *USENIX OSDI*, 2006, pp. 367–380.

[9] D. Ta, S. Zhou, X. Tang, W. Cai, and R. Ayani, "Qos aware server provisioning for distributed virtual environments," in *Proc. of ACM/IEEE/SCS PADS*, 2010, pp. 20–28.

[10] T. Henderson and S. Bhatti, "Networked games: a QoS-sensitive application for QoS-insensitive users?" in *Proc. of ACM SIGCOMM*, 2003, pp. 141–147.

[11] L. Pantel and L. Wolf, "On the impact of delay on real-time multiplayer games," in *Proc. of ACM NOSSDAV*, 2002, pp. 23–29.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[13] B. Zhang, E. Ng, A. Nandi, R. Riedi, P. Druschel, and G. Wang, "Measurement-based analysis, modeling, and synthesis of the internet delay space," in *Proc. of ACM SIGCOMM/USENIX Internet Measurement Conference*, 2006, pp. 85–98.

[14] D. Ta, T. Nguyen, S. Zhou, X. Tang, W. Cai, and R. Ayani, "A framework for performance evaluation of large-scale interactive distributed virtual environments," in *Proc. of IEEE International Conference on Scalable Computing and Communications*, 2010.

[15] B. Wong, A. Slivkins, and E. G. Sirer, "Meridian: A lightweight network location service without virtual coordinates," in *Proc. of SIGCOMM Conference*, 2005.

[16] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: Estimating Latency between Arbitrary Internet End Hosts," in *Proc. of ACM SIGCOMM IMW*, 2002.

[17] "http://pyxida.sourceforge.net," Retrieved on June 2010.

[18] W. C. Feng and W. C. Feng, "On the geographic distribution of online game servers and players," in *Proc. of NetGames*, 2003, pp. 173 – 179.

[19] C. D. Nguyen, F. Safaei, and P. Boustead, "Optimal assignment of distributed servers to virtual partitions for the provision of immersive voice communication in massively multiplayer games," *Computer Communications*, vol. 29, no. 9, pp. 1260–1270, 2006.

[20] K. W. Lee, B. J. Ko, and S. Calo, "Adaptive server selection for large scale interactive online games," *Computer Networks*, vol. 49, no. 1, pp. 84–102, 2005.

[21] C. E. B. Bezerra and C. F. R. Geyer, "A load balancing scheme for massively multiplayer online games," *Multimedia Tools Appl.*, vol. 45, no. 1-3, pp. 263–289, 2009.

[22] M. Lim and D. Lee, "A task-based load distribution scheme for multi-server-based distributed virtual environment systems," *Presence*, vol. 18, no. 1, pp. 16–38, 2009.

[23] J. Lui and M. Chan, "An efficient partitioning algorithm for distributed virtual environment systems," *IEEE Transaction on Parallel and Distributed Systems*, vol. 13(3), pp. 193–211, 2002.

[24] E. Cronin, S. Jamin, C. Danny, and R. Yuval, "Constrained mirror placement on the internet," *IEEE Journal on Selected Areas of Communication*, vol. 20, no. 7, pp. 1369–1382, 2002.

[25] L. Qiu, V. Padmanabhan, and G. Voelker, "On the placement of web server replicas," in *Proc. of IEEE INFOCOM*, 2001, pp. 1587–1596.

[26] S. Zhou, W. Cai, B. S. Lee, and S. J. Turner, "Time-space consistency in large-scale distributed virtual environments," *ACM Transactions on Modeling and Computer Simulation*, vol. 14(1), pp. 31–47, 2004.

[27] Z. Drezner, *Facility Location: A Survey of Applications and Methods*. Springer, 1995.